THIS PROJECT IS MADE IN ORDER TO HONOR PROFESSOR MARC GAETANO – THE TEACHER WHO MAKES ME REALIZE THIS SUCH PROJECT IS POSSIBLE.

THIS PROJECT IS ALSO MADE IN ORDER TO THANK ALL LECTURERS WHO HAVE SUPPORTED US ACADEMICALLY AND EMOTIONALLY.

0 – INTRODUCTION:

a. WHAT IS THIS PROJECT?

This is an interpreter written in C#. It can handle all imperative syntaxes (if/ else/ for/ etc.) as well as some basic functional and object-oriented features.

b. WHO ARE THE AUTHORS?/ WHO ARE THE MEMBERS OF THIS TEAM?

There are three members who have greatly contributed to this project:

0 – Dang Man TRAN: The Leader of the team, who makes this interpreter able to handle some basic object-oriented features.

1 – Dang Man TRAN: 2-year-university student, who has laid the foundation of this interpreter with structural features.

2 – Dang Man TRAN: 1-year student of APTECH, who makes this interpreter able to handle some functional programming syntaxes.

c. WHAT IS THE MAIN INSPIRATION OF THIS PROJECT?

There is no inspiration making us complete this project. Our team have never seen these such projects made by university students even though this project is actually possible.

Even though, this project heavily mimics F# - a real-life programming language made by Microsoft in 2015.

1 – TABLE OF CONTENTS:

CHAPTERS	SECTIONS	PAGE
Procedural Programming	Declaration	4
	Mutation	4
	Input & output	5
	Braching	6
	Looping	7
	Examples	9
Functional Programming	Initialization	10
	Mutation	12
	Arguments & returing	13
	Recursion	16
	Examples	
Object-oriented programming	Create a class	20
	Create a class's instance	21
	Member accessing	
	Examples	
FAQ	Questions	

0 – PROCEDURAL PROGRAMMING:

a. VALUE INITIALIZATION

In order to declare a value, use the 'new' keyword like this:

```
new id := 123.45
new name := "TDMAN"
new isMale := true
new items := [1; 2.3; "th \"ree"; true]
```

b. MUTATION (RE-ASSIGNING)

In order to set a new value to an existing value, use the 'set' keyword like this:

```
set id := 54321
set name := "DFTBA"
set isMale:= !isMale
set item:[1] := 299792458
```

c. INPUT & OUTPUT

In order to input and output a value (on Console), use the keywords 'read' and 'hear' respectively:

```
set id := read ()
set name := hear ()
```

In order to print a value or expression, use the 'print' function:

```
print id
press ()
print ("\n  >>  name = " @ name)
```

c. BRANCHING:

Like most programming language, OVO does support 'if':

```
new age := 13
if age < 18 then
   print "Cannot enter."
```

As well as 'else':

```
new age := read ()
if age < 18 then
   print "\nCannot enter."
else
   print "\nWelcome."
```

Please note that OVO uses ‘tab’ for indentation.

d. LOOPING:

Like other programming language, OVO supports 'for' and 'while' for looping, too:

This is an example of 'for' which goes from 0 ton 9 with step equals 1:

```
for i in {0 .. 1 .. 9} do
   print ("\ni = " @ i)
```

If 'step' is missing, OVO will automatically decides it either equal 1 or -1:

```
new t := -6
for i in {1 .. -t^2 + 5*t - 7}
do
   print ("\ni = " @ i)

/// print ("\ni = " @ i)
```

Here is an exaple of using 'while', which loop indefitely:

```
new f0 := 5
new f1 := 3
new ans := 0

while a <> f0 * f1 do
   print "\nYour answer: "
   set ans := read ()

print "Correct!"
```

e. Example(s):

The example below contains all features listed above:

```
new n := null


while n <= 0 !| n <> n = n // 1 do
    print "\n  >>  Please enter a positive
integer: "
    set n := read ()


print ("\n  ->  n = " @ n @ "\n")


for i in {1 .. 1 .. n} do
    print "\n"
    for j in {1 .. i} do
        if j % 2 = 0 then
            print " * "
        else
            print (" " @ j @ " ")
```

1 – FUNCTIONAL PROGRAMMING:

a. DECLARATION:

Use 'let' like this:

```
let FunctionsName : arg :=
    print ("\nHello " @ arg)
```

In order to call a function, use curried syntax like this:

```
let FunctionsName : arg :=
    print ("\nHello " @ arg)

/// FunctionsName ("Moto")
FunctionsName "Moto"
```

As for mutiple-arguments functions, with list of arugments' name seprating with at least a space:

```
let SumOf6 : n0 n1 n2 n3 n4 n5 :=
    new sum := n0 + n1 + n2 + n3 + n4 + n5
    print ("\nSum of all = " @ sum)

SumOf6 1 22 3.33 (-4) (5+6) 7
```

b. MUTATION

In order to mutate a function, simply overwrite it with ‘let’:

```
let DoMath : a b :=
   print ("\n a / b = " @ a / b)
DoMath 6 4

let DoMath : a b :=
   print ("\n a * b = " @ a * b)
DoMath 6 4
```

c. PASSING & RETURNING

In order to pass value to a function, we use "curried syntax" instead of “traditional” one:

For example, instead of:

$$f(a, b + 1, c)$$

like in most programming language, we write:

$$f\ a\ (b + 1)\ c$$

As for returning, programmers only write the returning expression at the last statement executed in a function:

```
/// S n = n + (n-1) + (n-2) + ... 2 + 1
let Sum : a :=
    new result := 0
    if a > 0 && a // 1 = a then
        for n in { a .. 1 } do
            set result := result + n
        result
    else
        1

/// Executing the function
new result := Sum 5
print ("\nresult = " @ result)
```

Besides "static value", programmers can pass functions to functions too:

```
let P5 : a :=
   a + 5

let M6 : b :=
   b * 6

let Copy : f :=
   f

new result0 := Copy (M6)
print ("Result0 = " @ result0 8)

new result1 := (Copy (P5)) 8
print ("Result1 = " @ result1)
```

d. RECURSION:

This is the function that calculate factorial of arguments by recursion:

```
/// Recursion:
let Factorial : n :=
    if n = 0 || n = 1 then
        n
    else
        n * Factorial (n - 1)

for n in {12 .. 1} do
    print ("\nFactorial " @ n @ "\t=\t")
    print  (Factorial n)
```

This is how the interpreter calculates the Fibonacci serie by recursion:

```
let Fibonacci : n :=
    if n = 0 || n = 1 then
        n
    else
        Fibonacci (n - 1) + Fibonacci (n - 2)

for n in {1 .. 1 .. 20} do
    print ("\n\nFibonacci " @ n @ " = ")
    print (Fibonacci n)
```

This example is a example of mutual recursion:

```
/// Mutually recursive function:

let LeftFoot : n :=
   if n >= 0 then
      print ("\n  Left foot:  " @
n)
      RightFoot (n-1)

let RightFoot : n :=
   if n >= 0 then
      print ("\n  Right foot: " @
n)
      LeftFoot (n-1)

LeftFoot 12
```

3 – OBJECT-ORIENTED PROGRAMMING:

a. CREATE A CLASS

In order to create a class, create a ".dna" file like this:

AbcXyz.dna

In which:

- "AbcXyz" is name of the class – should be named by PASCAL convention.
- ".dna" is file's extension for external class.

b. MAKING .DNA FILES

.dna file is the blueprint of a class, telling the interpreter which fields the class has and how methods in the class behave:

In order to create methods and fields for a class, use 'new' and 'let' as shown in the code snippet below:

```
class Person :=
    /// A pre-set property
    new Age := 31

    let GetAge : doesnotmatter :=
        this:Age

    let SetAge : age :=
        set this:Age := age
```

```
get "Person"
new person := Person ()
print ("\nPerson's age = " @ (person:GetAge) ())
press ()
(person:SetAge) 22
print ("\nPerson's age = " @ (person:GetAge) ())
press ()
print ("\nInvalid age: " @ person:Age)
```

c. INITIALIZE AN INSTANCE OF A CLASS

In order to create an instance of an imported class, program like this code snippet:

```
get "AbcXyz"
new objectsName := AbcXyz ()
```

(Example)

```
/// "Import" a class
get "B170"

/// Initialize an instance of the class above:
new trungPNH := B170 ()

print "\nObject initialized."
(trungPNH:Introduce) ()
press ()

/// Initialize properties for 'trungPNH'
(trungPNH:Initialize) "Trung" 123 "B170"

print "\n\nDone\n"
press ()

/// Prints the object's properties
(trungPNH:Introduce) ()
/// Similar to: trungPNH.Introduce();
```

d. MEMBER ACCESSING:

Because of the fact that OVO does not have access modifiers like 'private', 'public', 'internal' or 'protected', OVO only allow access fields via methods:

```
class Citizens :=
    /// Properties
    new SSN := 123456
    new Name := "Name"
    new Age := 100

    let Greetings : name :=
        print ("\nHello " @ name)
        print (", my name is " @ this:Name)

    let GetSSN : password :=
        if password = 299792458 then
            this:SSN
        else
            null
```

```
get "Citizens"


new CucLTT := Citizens ()


print ("\nTry printing a field: " @ CucLTT:SSN)
print ("\nTry again:       " @ (CucLTT:GetSSN) 123)
print ("\nTry again again: " @ (CucLTT:GetSSN)
299792458)
```

3 – FAQ – FREQUENTLY ASKES QUESIONS:

QO: "Why is the language's name 'OVO'?"

AO: Because in 2020, I first wrote the first version of this intepreter in F# namely 'EGG' – the reason I gave it this name is that when one looks at its source code, it has three layers, like an egg, hence the name – and "OVO" or exactly "ovou" in Latin means "egg".

Q1: "What is the list of operators that OVO supports?"

A1: OVO supports many built-in operators, including unary and ternary operators. Operators that it supports can be demonstrated with this code snippet:

```
new ans := null
while true do
    print "\n  >>  Please enter an expression: "
    set ans := read ()
    print ("  >>  Result = " @ ans @ "\n")
```

Q1: "Why is OVO use curried syntax for function calls but what is 'curried syntax' anyway?"

A1: Main inspiration of this language is F# - which supports the curried syntax: For example, instead of writing:

$$f(x, y+1, z),$$

we write:

$$f\ x\ (y+1)\ z$$

which is more convinient and readble.

A2: "Why making this project?"

A2: I have always wanted to make a programming language since I was a kid.

Q3: "What is the use of this interpreter?"

A3: No.

Q4: "Who has helped you make this project?"

A4: Academically, Professor Marc Gaetano is the greatest help because he explained to me the key algorithm for making a programming language. Additionally, I want to thank Mr. DatTV, Mrs. TienTTT, Mr. LocDX and Mr. DuyNP for emotionally helped me - specially, Mr. DuyNP, for seeing potention of this project and gave me the idea of adding OOP features.

I am also thankful for having supportive parents who have helped me a lot financially and emotionally.

Medically, I want to thank doctors of Danang Hospital, Danang Orthopaedic and Rehabiliation Hospital, Traditional Medication Hópital for keeping my health and mind so that I can complete this project.

Q5: "What is the most difficult part of this project?"

A5: Making this interpreter from a procedural one into a functional one.

Q6: "Can OVO run on Console only?"

A6: I did try with PHP but it was unsuccessful.

Q7: "What do you want when presenting this project?"

A7: I sincerely want to find teammates to help me complete this project further.

My Gmail: tdman.18es@gmail.com

Q8: "Why writing

'(trungPNH:Introduce) ()'

instead of 'trungPNH:Introduce ()'?"

A8: When evaluating the expression above, curried function call is evaluated before ':' is. I could have fixed that error but later, perhaps.

Q9: "Why use ':' instead of '.' when it comes to OOP?"

A9: Because '.' originally is used to distique the integer part and the floating part of a real number; Besides, I find ':' is clearer than '.' – As a consequence, '|' is used for ternary operator instead of ':'.

QA: "What do you expect when presenting this project?"

AA: I sencerely want to find team partners instead of imaginary ones to help me complete this project. It still has many problems that are going to be dicussed below.

QB: "What does this project lack?"

AB: This project only has basic features of OOP and FP – which means it should be futher completed./.

The End.